Comparing learning ecologies of primary graphical programming: create or fix?

Tom Neutens
IDLab, Department of Information Technology, Ghent University - imec, Ghent, Oost-Vlaanderen, 9000, Belgium

Evelien Barbion
Department of Educational studies, Ghent University, Ghent, Oost-Vlaanderen, 9000, Belgium

Kris Coolsaet
Department of Applied mathematics, computer science and statistics, Ghent University, Ghent, Oost-Vlaanderen, 9000, Belgium

Francis wyffels
IDLab, Department of Information Technology, Ghent University - imec, Ghent, Oost-Vlaanderen, 9000, Belgium

abstract

In the last few years, programming, computational thinking, and robotics are more frequently integrated into elementary education. This integration can be done in many different ways. However, it is still unclear which teaching methods work in which situations. To provide some clarity in this area, we compared two methods of integrating programming into a primary robotics workshop for learners aged ten to twelve. In one method, students create programs from scratch; in the other, they start with a faulty program they have to fix. These teaching methods were evaluated using the framework of learning ecology, which provides a holistic framework for assessing complex learning environments. We identified different indicators of learning ecology and assessed our workshops using a mixed-methods approach. Our results showed no difference between the groups on the intrinsic dimension of a learning ecology. However, on the experiential dimension, the learners in the create group scored better on all tests. Our results show the value of a multidimensional assessment of learning ecology to understand different teaching techniques. Additionally, the results provide us with important insights on how to integrate programming into a primary robotics curriculum enabling teachers to select better methods for teaching computing in their classroom.

Introduction

Primary school robotics covers a wide range of interrelated topics. One of these topics is computer programming. Integrating programming into a robotics course can be done

in a variety of ways (Kazakoff et al., 2013; Lee et al., 2014; Vasilopoulos & Van Schaik, 2019). However, many publications focus on exploring the effects of new teaching tools. Consequently, the analysis of teaching methods based on established theoretical constructs remains under-explored (Luxton-Reilly et al., 2018). One important theory often related to programming is cognitive load theory (Sweller, 2011). Cognitive load theory elucidates the relation between learning and the information processing system involving long-term and short-term memory. It is based on the premise that the human mind has two types of memory, a high capacity long-term memory which can more-or-less permanently store information, and limited capacity short-term memory, also called *working memory*, responsible for storing sensory information used for logical thinking and creativity (D. Shaffer et al., 2003). Cognitive load theory argues that, for effective learning to take place, the capacity of the working memory should not be exceeded. Working memory load can be affected by both the learning task itself (intrinsic cognitive load) as well as the way that the learning task is presented (extraneous cognitive load). Since these two types of cognitive load are additive, learning tasks with high intrinsic cognitive load should be presented in a way that limits extraneous cognitive load as much as possible (Van Merrienboer & Sweller, 2005). Certain types of instructional strategies like adding worked examples, reducing split-attention, adding redundancy, and increased modality have been shown to reduce cognitive load (Sweller, 2020). Getting insight into the best ways of mapping these strategies to teaching techniques for specific domains like computing requires further research. Moreover, determining which teaching techniques are effective in specific educational contexts is required to improve teaching practice. Malik and Coldwell-Neilson (2017) and Hsu et al. (2018) stress the importance of examining different learning strategies and analyzing how they relate to previously shown positive effects on computational thinking. Furthermore, Popat and Starkey (2019) state that exploring different teaching methods and identifying their effects on programming, as well as other educational outcomes, is an important topic for future research.

Generally, not unlike language learning, learning programming is split into reading and writing skills. Consequently, many proposed teaching techniques focus either on reading, writing, or a combination of both. Numerous introductory programming courses mostly expect learners to write code, also known as code generation (Van Merriënboer & De Croock, 1992). This teaching technique is often chosen since it is similar to what programmers do in real life. Mimicking real life should provide an authentic learning environment (D. W. Shaffer & Resnick, 1999). However, multiple studies have shown that this method has drawbacks. Van Merriënboer and De Croock (1992) have shown that students often had difficulties in finding a solution to the problems presented to them when having to generate code. Additionally, they showed that learners taught using the code generation method scored lower on a statements knowledge test, than students who were taught using code completion problems. Moreover, Garner (2009) has shown that teaching programming using the generation method required more time and triggered more questions from the learners than the part-complete solution method they describe. The part-complete solution method provides the learners with an incomplete program and requires them to add an element. Abdul-Rahman and Du Boulay (2014) compared this part-complete solution method to their proposed structure-emphasising method. The structure-emphasizing method requires learners to read code and give an explanation of what the code does. Their results show

no significant differences in terms of learning outcomes between the groups however, they show a plausible advantage of the code completion method in terms of learning efficiency.

While the generation method mostly focuses on code writing and the part-complete solution method focuses mostly on reading, other methods have been proposed which lie somewhere in between. An example of such a method is Parsons problems. These problems give the learners a set of blocks, each containing a part of the final program. The learners have to reorder these blocks to create a final solution (Du et al., 2020). Ericson et al. (2017) showed that solving two-dimensional Parsons problems with distractors (blocks which should not be added to the program) required less time than both writing the same code and fixing equivalent code with the same errors as the distractors. However, they did not show a statistically significant difference in knowledge retention one week after the experiment. With the more recent resurgence of programming in the last two years of primary school, other coding techniques have been created. One of the most prevalent is block-based programming (Weintrop & Wilensky, 2015). Block-based programming, not unlike code completion or Parsons problems, simplifies coding by reducing complexity. It favors recognition over recall, groups computational patterns into blocks, and prevents errors by limiting the syntactical changes learners can make (Bau et al., 2017).

Another strategy often used when teaching programming is making learners fix faulty programs (Liu et al., 2017). Finding errors in your program and correcting them is an essential skill programmers have to learn. Lee et al. (2014) have shown that teaching programming through debugging can lead to more success when learners create their own programs and possibly helps sustain learners' motivation. Additionally, Ahmadzadeh et al. (2005) argue that acquiring debugging skills increases a programmer's confidence. Moreover, studies have shown that teaching programming using this technique has a positive effect on problem-solving skills in general and facilitates transfer of these skills to other domains (Kim et al., 2018; Klahr & Carver, 1988; Witherspoon et al., 2017). Even though these studies have shown the benefits of teaching programming through fixing, the conclusions they draw are often speculative, based on limited empirical evidence. Consequently, further empirical evidence should be collected and analyzed to support these preliminary conclusions (McCauley et al., 2008).

Completion problems, Parsons problems, fixing code, and block programming all have their theoretical roots in cognitive load theory. The techniques described above aim to limit cognitive load by abstracting away the details of programming language syntax and grammar. Moreover, they reduce the number of options by grouping code into blocks. Often, these techniques are used in combination with a learning path that uses repeated examples and practice time to gradually introduce more complexities into the programs the learners have to create. In block programming and Parsons problems, cognitive load is reduced because these techniques limit syntax errors as well as constrain the problem space the learners have to work in (Ericson et al., 2017). Similarly, code completion problems limit the number of options learners have to choose from, reducing cognitive load. Learning through fixing provides learners with most of the code, narrowing down the problem to identifying the error, locating that error, and fixing it. It defines a clear procedure for solving problems as opposed to generating code from scratch where multiple strategies can lead to a solution. Having a clear strategy has been shown to reduce cognitive load (D. Shaffer et al., 2003).

In this paper we compare the learning outcomes and experiences associated with the code generation and code fixing strategies. Since these teaching strategies can influence different aspects of learning, we looked for a framework that enabled us to evaluate multiple relevant dimensions of learning in a structured way. Cobb et al. (2003) supports the decision for a multidimensional approach by stating that analyzing one aspect in a complex system limits our ability to establish theories about learning. Experiments should ideally result in a greater understanding of a learning ecology, which they define as a complex interacting system involving multiple elements of different types and levels. Jackson (2013) gives us a more extensive definition of a learning ecology. They identify five key components of an individual's learning ecology: contexts, relationships, process, will and capability, and resources. Luckin (2008) proposes a learning ecology of resources model. They define an ecology of resources as a set of inter-related resource elements, including people and objects, the interactions between which provide a particular context. They argue that different resources are needed when different subjects are taught and propose a framework of learning ecology in which different scaffolded exercises can be fitted. The model puts the learner in the center and states the interactions she has with the curriculum, knowledge, organization, environment, administration, and resources. The different definitions of learning ecology are synthesized by González-Sanmamed et al. (2019). In their paper, they use the Delphi method to create a general model of learning ecology based on previous definitions and expert knowledge. The model has intrinsic and experiential dimensions. The intrinsic dimension structures learning dispositions using conceptions, motivation, and expectations. The experiential dimension focuses on the learning process by analyzing relationships, resources, actions, and context. In our experiment, we used this framework of learning ecology as a basis for our assessment.

Research question

This study aims to provide some clarity about the benefits and drawbacks of teaching programming through fixing code for novice programmers aged 10 to 12 years old. To do this, we created two robotics workshops, one integrating programming using the traditional code generation method, the other using faulty programs which the learners had to fix. These workshops will further be referred to as the *create*- and *fix*-methods. The main research question we try to answer is: *Do teaching programming through fixing and teaching programming through creating have a different effect on learning ecology in the last two years of primary school?* Our research focuses on both learning achievement as well as learning experience which are both parts of the experiential dimension of learning ecology. To assess the learning achievement, we look at the acquisition of both programming skills as well as higher-level computational thinking skills. To assess learning experience we look at both the emotion during the workshop, use of programming and computational thinking concepts, as well as the interaction patterns with our programming environment. To get a sense of how the dispositions of the learners differ between the experimental groups we used an attitude test as a baseline measure.

Assessment methods

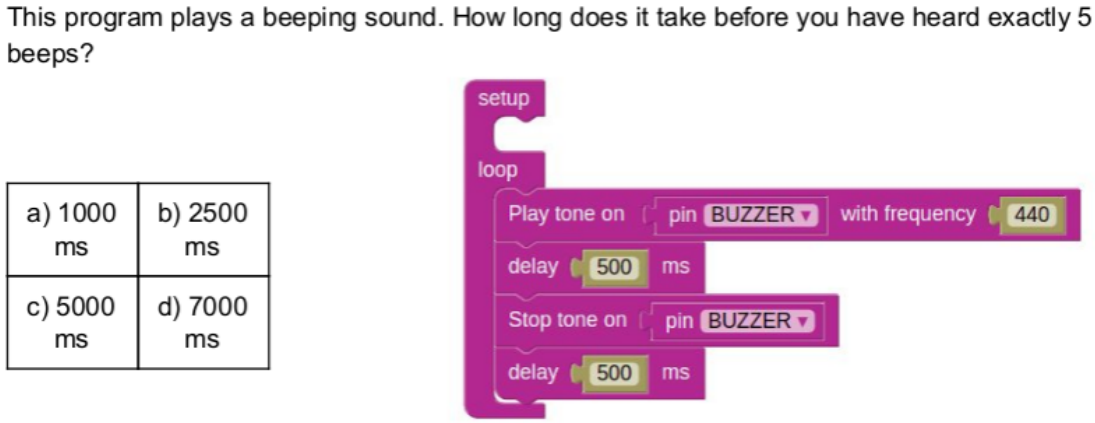
Our research team consisted of: One full-time researcher, a master thesis student, a teacher who could support us during some of the workshops, and a supervising professor. With this team, we aimed to assess both the learning dispositions and learning processes (González-Sanmamed et al., 2019) for both of the workshops we designed. To get a multi-dimensional perspective on the learning process, we combined quantitative with qualitative assessment. A wide range of assessment techniques was chosen to get a better perspective on the learning process. Based on our experimental goals, we selected the following metrics: (1) A quantitative assessment of attitude between test groups. (2) Quantitative assessment of the learning achievement for computational thinking and programming. (3) Qualitative assessment of programming, computational thinking, and emotion during the learning process. (4) A quantitative assessment of programming data. In the following paragraphs, we give more information about these assessment techniques and how they can be mapped to the model of learning ecology as well as our research question.

Attitude. To quantitatively assess the intrinsic dimension of our learning ecology, we selected the BRAINS test as a reference (Summers & Abd-El-Khalick, 2018). The test is designed to evaluate the attitude towards science of children aged 10 to 15 years old. The test includes 30 questions, which are split into five categories: attitude, intentions, beliefs, control, and norms. We translated the questions into dutch and replaced the term "science" with the term "science and technology", which is the term that is generally used in Flemish primary education to describe STEM subjects. We used an attitude test to assess the intrinsic dimension of learning ecology because it represents the dispositions learners have towards STEM. The BRAINS test specifically gives us information about the conceptions, motivations, and expectations the learners have towards science and technology. This test is not intended to compare the effect of our different workshops but to verify that there is no intrinsic bias towards the content of the workshop in one of the experimental groups.

Learning achievement. Since the fix method should limit extraneous cognitive load compared to the create method it should result in higher learning achievement (Abdul-Rahman & Du Boulay, 2014). Moreover, it should facilitate the acquisition of higher-order thinking skills (Klahr & Carver, 1988). Consequently, we created a set of programming questions as a direct measure for the skills taught during the workshop as well as a set of computational thinking questions to measure possible transfer to higher-order thinking. The set of programming questions measure if the learners understood the concepts taught during our workshop. These concepts are time behavior, iteration, selection, and condition. The questions were asked by showing the students a short piece of code and asking for the behavior this piece of code would exhibit; an example question is shown in Figure 1. To create a measure for the higher-order thinking skills, we first looked at the different definitions of computational thinking. Many definitions of CT exist: Lye and Koh (2014) describe it as a series of problem-solving skills and attitudes in which they use computer science concepts. Barendsen et al. (2015) define CT as a way to solve problems in a way they can be implemented as a computer algorithm. Hsu et al. (2018) synthesized the many definitions into a classification of CT. This classification defines:

- 19 skills, these are largely based on the ones proposed by Wing (2006) and include abstraction, algorithm design, and decomposition.

Figure 1. Example multiple choice question to measure basic programming concept knowledge.



- 16 common learning strategies like problem-based learning, collaborative learning, scaffolding, and teacher-centered lecturing.

- 14 tools for teaching CT, these range from older tools like LOGO (introduced by Feurzeig et al. (1969)) to more modern ones like LEGO and Scratch.

No standardized test exists for measuring computational thinking skills. However, Chen et al. (2017) describe a 23 item instrument to assess computational thinking in a robotics context. Moreover, Román-González et al. (2017) use a test consisting of three types of questions:

1. Computational concept questions. These assess if the students understand basic programming concepts like loops and variables.

2. Automated assessment using the Dr. Scratch environment.

3. Computational thinking concepts using Bebras questions.

The authors show a correlation between the scores on the basic programming questions and Bebras questions, which indicates that Bebras questions can be used to assess computational thinking. The Bebras questions are designed for the international Bebras competition and aim to measure the computational thinking skills of participants outside a computing context (Dagiene & Futschek, 2008). Each question tests one or more computational thinking concepts (Barendsen et al., 2015). Consequently, to test the computational thinking skills, we selected Bebras questions that test the concepts abstraction, decomposition, sequence, and algorithm design. We based our selection on previous classifications in literature (Barendsen et al., 2015; Dagiene & Sentance, 2016). Table 1 shows the questions used in the test using their official Bebras contest identifiers.

Table 1
Pre- and posttest Bebras questions

Question id	Name
2015-JP-04	Crane operating
2015-MY-01	Bracelet
2014-UA-05-1	Gold coins
2015-CZ-01	Chestnut animals
2015-JP-02	Animation
2013-SI-14	Waiter
2015-CZ-09	Height of animals
2014-SK-07	Bee robot
2014-LT-06	Robot stairs
2012-FR-06	Swapping

Experience. To get a deeper understanding of how the students experienced the learning process, we looked at when students used certain programming and computational thinking concepts or express certain emotions. As described in the introduction, the two teaching methods we compare could have a different effect on learning achievement. Analyzing how the learners experience programming and computational thinking should give us insight into the possible causes of those differences in learning achievement. Additionally, measuring emotion does not only provide us with a deeper insight into how the learners experience the workshop itself, it can also be related to cognitive load (Plass & Kalyuga, 2019). Giving us an extra data point for comparing our two teaching methods.

To analyze these three elements, we made video recordings of multiple learners during the sessions. To extract relevant information from the recordings, we defined three different coding schemes, one for computational thinking skills, one for programming skills, and one for emotion. The coding scheme for computational thinking is largely based on the work of Brennan and Resnick (2012) and Chen et al. (2017). We took their definitions of computational thinking and reduced the different elements they suggest to a set, which was relevant for us by eliminating the components which would not be used during the workshop we designed. Table 2 shows an overview of the coding scheme we used. We created a similar coding scheme that defines programming concepts. This coding scheme is mostly based on the coding constructs, which were used during the workshop. Table 3 shows the coding scheme that was used for programming concepts. Finally, we created a coding scheme to measure the emotions during the workshop. The coding scheme is based on the work of Bosch et al. (2013) who define 19 categories of emotion. According to Lewis et al. (2010) the emotions *fear* and *anxiety* are hard to differentiate during an observation. Consequently, these were combined into one concept (fear) for our experiment. We compared their classification with the one proposed by Rodrigo and Baker (2009). However, all the categories they proposed were already included in our coding scheme. We added the categories *pride*, *humor* and *affection* as proposed by Else-Quest et al. (2008). To simplify the classification of emotions we decided to remove the categories *uncertainty* and *helplessness* since these mostly result in *anger* or *sadness*. Finally, we added the

Table 2
Coding scheme used to encode the computational thinking concepts.

Code	Category	Description
1	Syntax	The learners use the right programming blocks and grammar defined by the environment.
2	Data	The learners use the right data and datatypes.
3	Algorithms	The learners use an algorithm that solves their problem.
4	Efficiency and effectivity	The learners solve a problem in an efficient way.
5	Incremental and iterative	The learners repeat some steps if necessary and choose to adapt their way of solving the problem.
6	Abstraction	The learners remove unnecessary information to get a simplified representation of the problem.
7	Modularisation	The learners get to a solution by combining solutions to partial problems.
8	Testing and debugging	The learners use a trial and error technique to resolve errors.
9	Reuse	The learners reuse previous solutions.
10	Decomposition	The learners split the problem into sub-problems.
11	Logic	The learners use if-then reasoning.
12	Generalisation	The learners create a rule or theory based on the patterns they observed.
13	Pattern recognition	The learners recognize repetitive patterns when solving a problem.
14	None	No computational thinking skills are used.

emotion *engagement* to the coding scheme since it was of interest to us. This resulted in the coding scheme shown in table 4. For each of these coding schemes, we transcribed different fragments to illustrate the interactions when some of the elements in the coding scheme occur. This qualitative data gives us an understanding of the experiential dimension of the learning process. It clarifies the relationships between learners, the actions they take, and the context within which they learn.

Code interaction. The final measure we used is the interactions learners had with the programming environment. Since we created the programming environment ourselves, we were able to log all interactions learners had with the environment. We performed a quantitative analysis of these interactions by counting how many times each of the following interactions occurred: total number of code changes, number of programs containing a conditional statement, number of programs containing a loop statement, number of programs containing a wait instruction. This logging data gives us more insight into the actions learners take and how they use the available resources resulting in more information about the experiential dimension of a learning ecology.

Table 3
Coding scheme used to encode the programming concepts.

Code	Category	Description
1	For-loops	An instruction to repeat a certain set of instructions a number of times
2	Debugging	Finding errors in the program using step by step debugging.
3	Conditions	A statement which is evaluated to true or false.
4	Repetition	Repetition of a set of instructions.
5	Sequence	Putting a set of instructions in the right order.
6	Wait	Using time to change the behaviour of a program.
7	Setup-block	Startup block used to setup the environment.
8	Loop-block	Startup block used to repeat a specific set of code blocks.
9	Clear-lcd-block	Clear the lcd-screen
10	DC-motor-block	Controls the speed of a specific motor.
11	Play tone-block	Plays a tone on the buzzer.
12	Sonar-block	Contains the distance measured by a sonar sensor.
13	Led-block	Turns a certain LED on or off.
14	If-then-else-block	Block used to check a conditional statement and execute the relevant result.
15	None	None of the specified coding concepts were used during programming.

Method

Our experiment was set up so the results should be easily transferable to the current educational context. We believe our research should contribute practical information for teachers allowing them to improve their teaching practice. Consequently, we took the educational context in which teachers in the region of Flanders (Belgium) work today as a reference for our experiment. Using this context as a reference, we identified multiple constraints that both the instructional and experimental design had to meet. We identified constraints enforced by both practice and policy. The first major constraint in primary school is time. A governmental policy defines ten topics primary schools have to teach (Vlaanderen, 1997). From these ten topics, one focuses on science. Within the science topic, only a limited number of educational goals focus on technical systems and processes. Consequently, in practice, teachers look for content that covers a wide range of educational goals within a limited time. The second constraint teachers often face, is their limited set of resources. Most of the budget provided by the government is used to pay teacher salaries (Vlaanderen, 2019). This leaves little resources to invest in teaching materials, especially for subjects that represent only a small part of the curriculum. The third and final constraint we identified is teacher content knowledge. Primary school teachers have minimal knowledge about more complex STEM topics since they were not educated to teach them. Consequently, teachers often depend on external partners to help them bring more complex content into the classroom. We took these constraints into account when creating

Table 4
Coding scheme used to encode emotions.

Code	Category	Markers
1	Anger	Inner eyebrows pointing downward, heightened upper eyelids, lips pressed against each-other, heightened upper lip.
2	Contempt	Using a sarcastic tone of voice, voicing remarks, lifting one side of the lips.
3	Boredom	Looking around, moving around in their seat, fiddling with something, resting their chin in the palm of their hand, explicitly stating: "this is boring".
4	Confusion	Scratching their head, repetitively looking at the same things, having a discussion with teacher or peers: "Why is this not working?"
5	Engagement	Focus on the assignment, relaxed face and body, focused on the task, listing possible solutions to themselves, peers or computer.
6	Happiness	High-fives, laughter, clapping their hands, fist in the air, "Wow!", "Cool!", "Yes!".
7	Fear	Tense body, tense mouth, raised eyebrows.
8	Frustration	Outburst, sharp movements, slamming keyboard or mouse, cursing, less intense facial expressions than seen with anger.
9	Sadness	Corners of the mouth face downward, inner eyebrows up, eyelids down, crying.
10	Surprised	Suddenly sitting upright, gasping for air, expressions like: "Huh?", "Oh".
11	Affection	Soft voice, giving compliments, physical attention.
12	Humor	Smiling, laughing, more intense expressions than category 6.
13	Pride	Sitting up straight, showing off, similar expressions as category 6.
14	Neutral	n.a.

our instructional and experimental designs. In the following sections, we describe both of these designs in more detail and explain how we met the constraints listed above.

Experimental group

Since STEM education is getting a more prevalent role in primary education in Flanders, our experiment focuses on the last two years of primary school. The learners who participated in our experiment were between the ages of ten and twelve. They were randomly selected from 10 different primary schools in Flanders (the Dutch speaking part of Belgium). Since we selected a random sample from Flemish primary education, our experimental group (N=211) reflects the underlying cultural and gender distributions present in primary education. Our results show our final group had 53% girls and 46% boys 1% preferred not to identify their gender. About 70% of participants indicated speaking mostly Flemish at home while about 20% sporadically or never speaks Flemish at home. These distributions match the ones for all of primary school provided by the Flemish government [1].

Instructional design

We combined the constraints described above with the requirements dictated by our research questions to create an authentic learning path containing multiple different learning experiences. The learning path consists of three workshop sessions of about 150 minutes, resulting in 7 hours and 30 minutes of learning. The workshops took place in the classroom of the students. The teachers of the classes were always present to help where possible and to help keep order in the classroom. However, the workshops themselves were given by either a researcher or a supporting teacher who was part of the research team. Using an expert teacher reflects how STEM is often integrated in primary school today. Since primary school teachers lack the necessary content knowledge, they often rely on external partners to fulfill some parts of the curriculum. When selecting content for the educational design, we took into account the governmental educational goals and aimed to match the content with a selection of these goals. After careful consideration, we created a physical computing learning path that seamlessly integrates the governmental educational goals about technical systems and processes. To test our hypotheses, we created two variations of the learning path in which we change the way the learners program the physical system. The first variation focuses on creating programs from scratch allowing learners to build their program step by step. This learning path incrementally introduces new programming concepts by explaining them using an example and asking the students to solve one or more exercises using that concept. The exercises require the learners to write a program from scratch trying to achieve the functionality we requested. The second variation uses a learning approach that focuses on changing and fixing programs. In this setup, similar to the *create* workshop, the learners first get a short explanation of a specific coding concept using an example. After each concept is introduced, the learners perform one or more exercises on that concept. In the exercises, they get a faulty or incomplete program and have to change the program to reach their goal. To keep the focus on a specific concept, the learners only have to either change one block in the program or add one block to the program. Moreover, during the

[1] https://onderwijs.vlaanderen.be/nl/nl/onderwijsstatistieken/statistisch-jaarboek/statistisch-jaarboek-van-het-vlaams-onderwijs-2018-2019

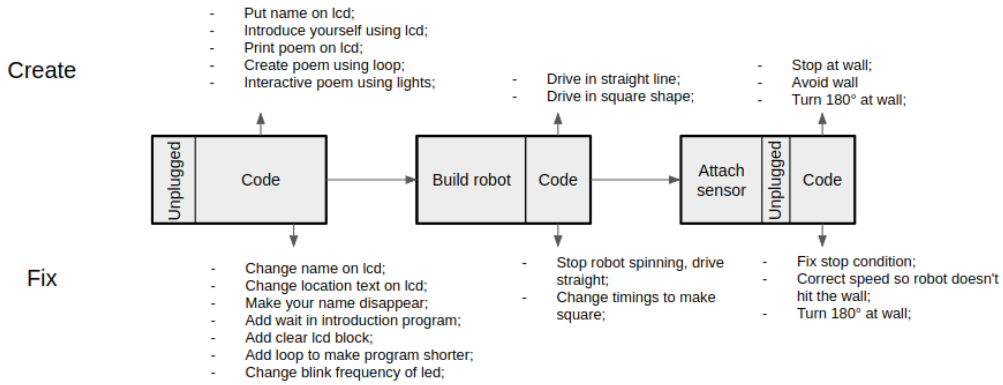
first session, the learners are instructed to write down the answers to the following questions: (1) What should the program do? (2) What does the program do now? (3) At what point in the program does it go wrong? (4) How can we fix the error? These questions provide a strategy for tackling the problems facilitating the learning process. In the second and third session we did not explicitly ask the learners to answer these questions enabling them to choose the solution strategy they wanted.

Figure 2 shows an overview of the learning path with the variations labeled as *create* and *fix*. All building and programming activities were done in groups of two. We paired up the students for multiple reasons. Firstly, a robotics workshop requires a lot of hardware including a robot kit with mechanical and electrical components as well as a laptop. Classrooms simply don't have the physical space to provide each learner with a full setup. Secondly, previous research suggests pair programming is advantageous for the learning process. Zhong et al. (2016) have shown that learning to program in pairs did not affect achievement and confidence among groups but did make girls more productive and confident. Moreover, Celepkolu and Boyer (2018) have shown that programming in pairs exposes learners to different ideas, reduced their frustrations, and helped them form social connections. Additionally, McDowell et al. (2003) have shown that programming in pairs improves pass rates and retention. Because literature shows working in pairs has mostly benefits and limited shortcomings, we found it appropriate for our experiment. Thirdly, having students work together will reveal more of the thinking patterns in our video recordings since communicating their thoughts is essential when working together. Consequently, it allowed us to get insights into their thinking patterns through the social interactions during the learning process.

Each workshop session contains at least one hour of programming exercises. The content of these exercises is different between the *fix* and *create* workshops. A detailed list of the exercises for each track is also shown in Figure 2. In addition to the coding sessions, the workshop contains two CSUnplugged activities (Bell et al., 2009). These familiarize the students with a computational concept which they are going to need during the exercises that follow. The first Unplugged activity familiarizes the learners with the concept of programming by making them write a program that lets their teacher make a sandwich. The second activity introduces the concept of conditional statements by making the students perform a certain act (for example: jump in the air) when the condition shown on the blackboard is true for them. Even though the *fix* and *create* workshops use different exercises, their objectives are the same. An overview of the learning objectives for the sessions is shown in List A.

To reach these learning goals, we created a custom set of teaching tools specifically designed for our workshop. These teaching tools include: (1) A custom-designed Arduino-based microcontroller board for robotics in education. The main features of the board include: An LCD-screen, 9 LED-lights, a buzzer, five buttons, and a built in motor driver to control different types of motors. The board enables learners to connect and control the components of a basic robot easily. (2) A custom robotics kit, designed to be inexpensive and easy to produce in a maker lab. It includes two inexpensive DC-motors with wheels, a set of laser-cut parts to construct a frame, a sonar sensor, and a set of standard screws. (3) A tailor-made open-source graphical programming environment based on Google Blockly

Figure 2. Graphical overview of the instructional design for the intervention. In the center, a high-level description of each session is given. Above and below a list of programming exercises for the *create* and *fix* group are given respectively.



[2], which includes a robotic simulator and debugger. Figure 3 shows a screen-shot of the environment. It has the standard elements like the Google Blockly toolbox and workspace but extends on it in different ways. The most important extension is the simulator view. This view provides the learners with a simulated environment in which they can test their code. All the tools are open source and designed to be inexpensive so they can be used in all schools, even with a limited budget for STEM education.

Experimental design

Our experimental design aims to identify the differences between the two test groups (*create* and *fix*). The design can be split in two main parts: quantitative assessment and qualitative assessment. As discussed in the assessment section on page 5, the quantitative assessment was done through a programming test, a computational thinking test, an attitude survey, and the logging of interactions with the environment. The data logging was done during all workshops. The attitude test was administered after the workshop together with the programming and computational thinking tests, which were presented to the students as one knowledge test. Note that, in the results section, we show the results for two different programming tests and two different computational thinking tests. The reason we have two tests is that we initially opted for a pre-posttest design where pre- and posttest are flipped between groups. However, after analyzing the results for the pretest, we noticed a low internal consistency score ($\alpha = 0.53$) on the programming test, indicating the test was filled out more or less randomly by the learners because they had little to no knowledge about the subject. Consequently, we opted to compare the groups solely based on the results on the posttest. Table 5 shows an overview of the participant distribution. The students in this distribution come from 10 different class groups from 7 different schools.

[2] https://developers.google.com/blockly

Figure 3. Overview of the custom programming environment. (1) The toolbox repository with all the blocks that can be used. (2) The workspace area, this is where programs are constructed. (3) The microcontroller board simulation. (4) A simulation of a riding robot similar to the one they build during the sessions. (5) Controls for saving, restoring and uploading the code to the microcontroller board

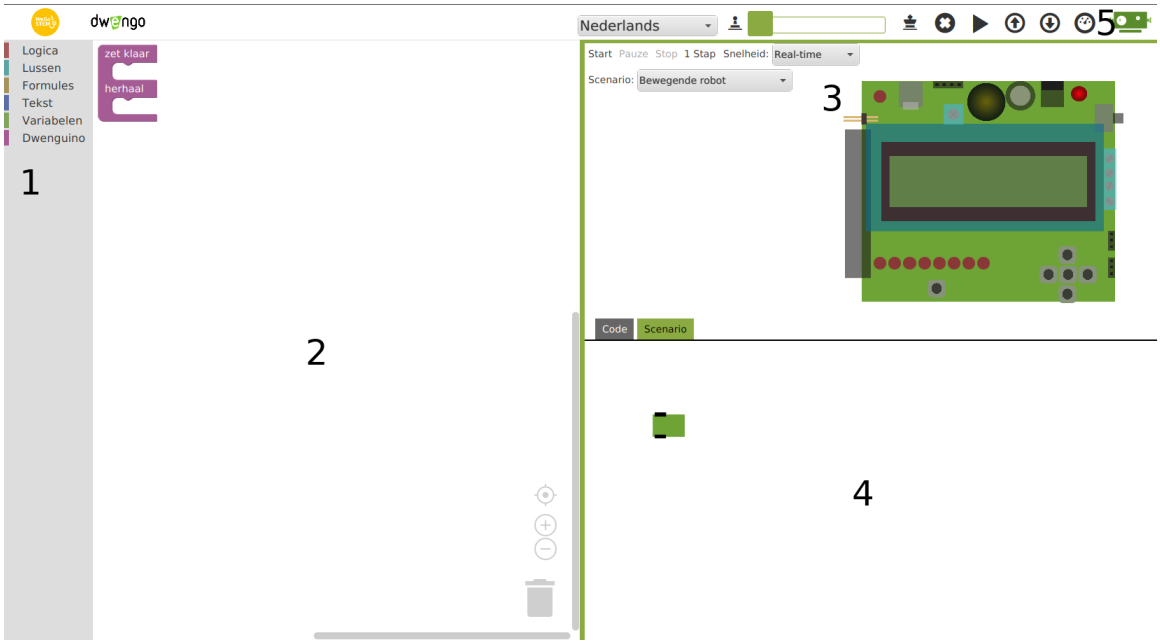


Table 5
Overview of participants distribution. (N=211) Since our initial experimental setup aimed to measure the learning progress on a pre-and posttest, we designed two tests. Consequently, Test 1 and Test 2 have different but topic matched questions. For half of the groups test 1 was used as a pretest and test 2 as a posttest. However, since the results on the pretest were not internally consistent, we decided to only compare the groups using the posttest. Note that the test groups presented here vary in size, this is because we had to remove some of the groups from our analysis since teachers did not administer the test correctly rendering it invalid.

	Test 1	**Test 2**	**Total**
Create	36	47	83
Fix	87	41	128
Total	123	88	211

For our qualitative assessment, we opted for video recording of specific learners during the workshop. According to Koch and Zumbach (2002) this is one of the methods which results in thorough encoding. Concretely, we randomly selected four student duos from which we recorded video and audio data during the full workshop[3]. These four duos came from both experimental groups, two in the *create* group, and two in the *fix* group. These recordings were analyzed using the three coding schemes discussed in the assessment section on page 5. To encode the data streams, we split them into sections of 30 seconds. To each section, we assigned three codes, one for computational thinking skills, one for programming skills, and one for emotion. Additionally, we selected specific interactions between learners as examples to support the findings from the coding process.

Results

To analyze our results, we refer back to the model of learning ecology by González-Sanmamed et al. (2019). We start by analyzing the differences between the *create* and *fix* groups on the intrinsic dimension. Afterward, we analyze the differences in the experiential dimension. Finally, we combine both dimensions to get an overview of the learning ecology in both groups.[4]

Intrinsic dimension

The intrinsic dimension is shaped by inter-subjective elements that characterize the self. It is mainly influenced by an individual's life trajectory, which includes different learning experiences. To understand the learning dispositions in the intrinsic dimension, we first describe the four duos whom we recorded during the workshops. The goal is to understand the affinity the learners have with programming.

Duo 1 (create) Mary and Susan: Mary and Susan came into contact with programming only once before during a small workshop. According to the teacher, they had not done any programming this year. Neither indicated that they had done any STEM activities outside of school.

Duo 2 (create) James and Lisa: James and Lisa indicated they had programmed before using Minecraft. James had more knowledge about STEM than the rest of the class because he won a local STEM competition that year. He told us he also had to program during that competition. Additionally, during one of the sessions, he said he had a programmable robot with sensors at home.

Duo 3 (fix) David and Mark: David and Mark had some experience in programming. They had multiple programming sessions at school this year using Minecraft and Scratch. David said he had an Arduino at home, which he used to program a traffic light.

Duo 4 (fix) Ruth and Sarah: Ruth and Sarah have some programming experience. During the instruction, they imply having programmed in Minecraft before. This is illustrated when the teacher asks them to think of a condition which can be true or false, they suggest: "If there is lava, put down a block".

[3]We obtained active, informed consent from the parents of learners who were videotaped. The other students in the group received a passive consent form explaining the anonymous data we collected.

[4]The data that support the findings of this study are available on request from the corresponding author. The data are not publicly available due to privacy or ethical restrictions.

Table 6
Average attitude score for each test group per dimension (Attitude, Intentions, Beliefs, Control, Norms). Significance of the differences between test groups was calculated using t-test.

	Attitude	**Intentions**	**Belief**	**Control**	**Norms**	**Total**
Create	3.220	2.577	3.231	3.280	2.402	3.025
Fix	3.081	2.456	3.269	3.281	2.400	2.984
p-value	0.426	0.368	0.669	0.995	0.967	0.686
Significance	None	None	None	None	None	None

From these descriptions, we can identify minor dispositions some of the learners have. Some learners, like James and David, are clearly intrinsically motivated for STEM and programming activities since they had voluntarily done it at home. Moreover, all the learners seem to have at least some experience with programming. However, this experience is limited to some irregular workshops and not part of the curriculum. Apart from some minor differences, the duos seem to have similar learning dispositions. To validate this observation, we compared the attitude towards science and technology between the *create* and *fix* groups. As described in the assessment section on page 5, all learners (N=211) filled out an attitude questionnaire assessing the dimensions: attitude, intentions, beliefs, control, and norms. An overview of the results is shown in table 6. The table clearly shows no significant differences between the groups on any of the dimensions. Consequently, both groups have equal dispositions, which imply there was no intrinsic bias about science and technology in each of the groups.

Experiential dimension

Before describing the results of the coding schemes defined in the assessment section on page 5, we provide a brief description of the learning experiences of each duo we followed.

Duo 1 (create) Mary and Susan: Mary and Susan quickly solved the first programming assignments. Generally, they solved the assignments more quickly than the other students in the group. Building the robot frame went smoothly. Mary indicates that she is not dexterous when it comes to building the robot. Both of them were focused during the workshop. However, when they got the opportunity to experiment, they acted more playful. Nevertheless, in those moments, they were always focused on the robot without being distracted. During the last session, Mary started losing focus from time to time. She started talking to other groups and asked for help more quickly, but only when they had been stuck for a while. Even though Mary was more distracted, Susan kept her focus and kept trying to solve the problems. When the duo finished an assignment more quickly than the other children in the group, they voluntarily started helping other groups. Mary and Susan were proud when they found a solution, and they wanted to share their solutions with other groups in the classroom and showed the result to the camera that was recording them. Collaboration between the duo went without issues. When they had a different method in mind, they talked with each other to get to the best solution.

Duo 2 (create) James and Lisa: Since James had much knowledge about programming, the duo quickly solved the programming assignments. At the end of the final

session, the instructor had to give them an extra assignment because they had finished all the exercises before all other groups. James and Lisa collaborated very well. James took the responsibility of the programming while Lisa spent her time building and modifying the robot. The duo was calm during the workshop. They were not playful or loud. During the moments of instruction, they were focused and listened carefully. When they finished an assignment early, they quietly waited for a new assignment. James was very focused during all workshops, and this was articulated in the way he worked. While programming, he always talked to himself, describing what he was doing and how he tried to solve specific problems. He was also very self-demanding when the duo had to program the robot to ride a square pattern on the floor, both Lisa and the teacher found their solution acceptable. However, James kept going because he wanted it to be a perfect square. When testing his code on the real robot, James sometimes got frustrated because it did not do what he expected. The real world often introduces unexpected conditions like different friction on each of the wheels of the robot. Consequently, theoretically correct solutions do not show the desired behavior in the real world. James clearly had not experienced problems like these before. Lisa spent most of her time building the robot and testing the code James had written. She had limited input in the programming process an often asked James if he knew the solution. She was more inclined to ask for help when they did not immediately find the solution to a problem. Especially in the final session, Lisa often got impatient and did not take the time to think about how to solve the problem. James and Lisa were proud when they solved the problem, and they even showed one of their solutions to the ICT-teacher who was in the room next to the one of the workshop.

Duo 3 (fix) David and Mark: David and Mark were able to solve most assignments during the workshop. However, they did experience difficulties during some of the exercises. For example, in one of the exercises, they had to identify why the LCD-screen did not show a word that was present in the code and add a wait-block to solve the problem. The duo identified the problem but was not able to add the correct code block to solve it. When David finally found the solution, his animated reaction demonstrated his excitement. When the duo was trying to make the robot stop at a wall using a sensor, they were stuck. After trying to correct the error using trial-and-error for 10 minutes, they asked the teacher for help. The problem seemed to be that they did not know what the smaller or equal sign meant. Once the teacher explained the sign, they were able to solve the problem. While building the robot, they did not experience any issues. Both took turns constructing part of the robot, David let Mark take over the construction when they had to tighten the small screws because he was bad at it. David and Mark were mostly focused during the workshop. Only when they finished an assignment faster than the other groups, David left his seat and started interacting with other groups. After a few minutes, Mark went looking for him and brought him back to help with the next assignment. The duo collaborated very well during programming, one was controlling the computer while the other helped thinking, and they often voluntarily switched places.

Duo 4 (fix) Ruth and Sarah: Ruth and Sarah quickly familiarized themselves with the programming environment. However, they did have some troubles during the workshop. When building the robot, they had difficulties attaching some of the screws. They often sighed, and Sarah cursed while trying to attach the motor. When they had to correct a program that should stop at the wall, they spent much time looking for the problem but

were unable to find a solution. After they got help, they found out their sensor was malfunctioning. Sarah got frustrated and looked sad. She lost all motivation and stopped actively participating in the process. Ruth did not give up, and she kept looking for a solution to the problem. Most of the time, the duo kept the focus on the assignments. When they finished an assignment before the other groups, Ruth suggested to try and build their own program. Ruth and Sarah collaborated very well. Most of the time, they helped each other to reach a solution. However, at a certain point when building the robot, they disagreed about how the sensor should be attached, which resulted in an angry reaction of Sarah: "If you know better, you do it!".

These four descriptions provide some insight into the learning process. Even from these small summaries, we can deduce that the learners in the *fix* group had more difficulties solving the assignments than the ones in the *create* group. These difficulties sometimes resulted in more anger and frustration during the workshop, which led to a loss in motivation for some learners. To get a more in-depth insight into the differences between the two groups, we apply the coding schemes described in the assessment section on page 5. Table 7 shows a list of programming concepts and the number of time steps (=30sec). The learners talked about this concept when programming. These results reveal some interesting insights, first and foremost, there seems to be a big difference in the number of concept mentions between the *create* and the *fix* groups. The *create* group talked more about the selected programming concepts. A second insight this data demonstrates is how little some coding concepts are used, especially in the *fix* group. It makes sense that the *create* group uses more of the programming concepts since none of the code is provided to them. However, in the *fix* group, most of the concepts are only mentioned a few times, indicating less meaningful interaction with the code. This discrepancy could be explained by the process itself. In the *create* group the learners have to select the blocks they are going to use in their program. This selection process sparks discussion about the different blocks they should use. In the *fix* group, much time is spent identifying the error in a program. Once the fault has been identified the learners try to match the code where they think the error is with structures of blocks they have seen before or are present in the code currently on the screen. If they find a similar structure they match it at the place where they think the error is without thinking about the specific coding concepts present. When this technique does not work, the learners mostly try to solve the problem using trial-and-error by modifying the code. This trial-and-error process puts less focus on understanding the code and more on identifying how the output changes when a specific element of the code is changed. Often the students find the solution just by spending a lot of time trying different things instead of finding a solution by understanding how the program works.

To continue our analysis, we investigated the interactions using the computational thinking coding scheme explained in section 2. Table 8 shows the number of time slots the learners use one of the specified computational thinking concepts. As with the programming concepts, the learners in the *create* group generally use more of the computational thinking concepts than the *fix* group. Remarkably, the *create* group uses the concept *testing and debugging* more often than the *fix* group. The only way we can explain this difference is that the learners in the *create* group spend less time reading through the code. They can start writing small parts of the program and testing them. That way they incrementally

Table 7
Number of lesson phases in which students talk about a certain programming concept. Group one and two participated in the create workshop, group three and four participated in the fix workshop. The significance is shown just as an indication, and we do not believe it has much value on a small group like this.

Concepts	**Group 1 and 2**	**Group 3 and 4**	**Both**	X^2 **(p-value)**
For-loop	7	0	7	5.489(.019)
Debugging	0	1	1	1.29(.256)
Conditions	6	4	10	0.057(.812)
Repetition	2	3	5	0.543(.461)
Sequence	8	7	15	0.055(.815)
Wait-block	33	4	37	17.17(.000034)*
Setup-block	2	4	6	1.297(.255)
Loop-block	8	1	9	3.936(.047)
Clear-lcd-block	20	6	26	4.669(.031)
DC-motor-block	7	2	3	4.500(.034)
Play-tone-block	2	1	3	0.132(.717)
Sonar-block	4	1	5	1.150(.284)
Led-block	5	1	6	1.798(.180)
If-then-else-block	5	0	5	3.910(.048)
None	289	274	563	15.759(.000072)*

* Bonferroni-corrected significance $p < 0.05/15$

construct their programs. The students in the *fix* group spend a lot of time just executing a program to see what it does. They need more time before they can start identifying the problem and testing possible solutions.

The emotions learners had during the workshop are shown in table 9. Again there is a noticeable difference between the *create* and *fix* groups. The students in the *create* group showed more engagement while the students in the *fix* group were more surprised. The *fix* group seems to have more negative emotions like anger, contempt, boredom, frustration, and sadness than the *create* group. The differences in each of these emotions are not significant. However, when looking at all negative emotions, we do observe a general trend.

Now that we have established the differences between the small subgroup of learners in our qualitative assessment, we look at our knowledge test if these results match the established trend. As stated before, both the *create* and *fix* groups were each split into two test groups. One of the test groups filled out test 1 the other test 2. Consequently, about half of the learners in the *create* group answered the questions in test 1, the other half answered the questions in test 2. Similarly, about half of the learners in the *fix* group answered the questions in test 1 while the other half answered the questions in test 2. As described in the assessment section on page 5, the test had both programming questions and Bebras questions. We checked the internal consistency of all tests using Cronbach's alpha, and this resulted in an internal consistency score greater than 0.95 for all tests. As shown in table 10, the *create* group scores consistently higher than the *fix* group on all tests. For both programming tests, this difference is significant, with a p-value less than 0.05. The

Table 8
Number of lesson phases in which students talk about certain computational thinking concepts. Group one and two participated in the create workshop, group three and four participated in the fix workshop. The significance is shown just as an indication, and we do not believe it has much value on a small group like this.

Concepts	**Group 1 and 2**	**Group 3 and 4**	**Both**	X^2 **(p-value)**
Syntax	21	1	22	14.153(.000169)*
Data	20	17	37	0.080(.778)
Efficiency and effectivity	28	2	30	17.469(.000029)*
Incremental and iterative	22	10	32	2.114(.146)
Abstraction	14	8	22	0.498(.481)
Testing and debugging	170	137	307	0.187(.666)
Decomposition	5	3	8	0.127(.722)
Logic	22	4	26	8.800(.003)*
Generalisation	0	1	6	4.698(.265)
Pattern recognition	6	0	6	4.698(.030)
None	90	126	216	27.048(<.0001)*

* Bonferroni-corrected significance $p < 0.05/11$

Table 9
Number of lesson phases in which students express certain emotions. Group one and two participated in the create workshop, group three and four participated in the fix workshop. The significance is shown just as an indication, and we do not believe it has much value on a small group like this.

Emotions	**Group 1 and 2**	**Group 3 and 4**	**Both**	X^2 **(p-value)**
Anger	0	1	1	1.290(.256)
Contempt	0	2	2	2.583(.108)
Boredom	0	1	1	1.290(.256)
Confusion	34	36	70	1.883(.170)
Engagement	160	64	224	30.524(<.0001)*
Happiness	47	47	94	1.746(.186)
Fear	1	0	1	0.777(.378)
Frustration	6	10	16	2.350(.125)
Sadness	2	4	6	1.297(.255)
Surprised	4	14	18	8.715(.003)*
Affection	1	4	5	2.696(.101)
Humor	7	9	16	1.047(.306)
Pride	12	7	19	0.374(.541)
Neutral	124	110	234	1.551(.213)

* Bonferroni-corrected significance $p < 0.05/14$

Table 10
Average score on each of the knowledge tests with the significance of the difference between the two test groups using a t-test.

	Test 1		**Test2**	
	Create (N=36)	**Fix (N=87)**	**Create (N=47)**	**Fix (N=41)**
Programming	2.389	1.885	1.680	1.073
	p = 0.034		0.006	
	Create (N=36)	**Fix (N=87)**	**Create (N=47)**	**Fix (N=41)**
Bebras	2.610	2.340	1.550	1.366
	p=0.286		p=0.430	

differences in the Bebras questions are not statistically significant. However, there does seem to be a general trend indicating a lower performance of *fix* group.

The final dimension we analyzed is the logging data collected from our programming environment. During the workshops, we saved all interactions learners had with the programming environment. It includes each time the learners changed an aspect of the code, and each time a new block was created by dragging it from the toolbox. To compare the concepts condition/selection, for-loops and time-delay, we looked at the percentage of code changes that were done on code containing one of these concepts. This can be interpreted as the percentage of the time in which that concept was present in the learners' code. Table 11 shows an overview of the results. Clearly, the *create* group used a lot more blocks from the toolbox than the *fix* group. Additionally, the *create* group interacted more with the code, which is demonstrated by the number of code changes. Interestingly, even though the *create* group interacted more with the code than the *fix* group, the relative occurrence of the programming concepts varies across the groups. In the following paragraph, we give a possible explanation for these differences.

Condition/selection is more present in the *fix* group. Since the exercises in the *fix* group never required the learners to add a conditional statement, only to modify existing conditional statements, this percentage can be interpreted as the time learners spent on the challenges containing a selection statement. In the *create* group, the learners had to understand when and where to use the selection statement. Once they understood, less work was required to finish the challenge. In the *fix* group, the learners got a program with a selection statement. Consequently, before they understood what the statement meant, they interacted with the code to figure out the problem and fix it. The same reasoning can be used to explain the difference in occurrences of time-delays. However, for-loop are used a lot more often by the *create* group than by the *fix* group. This is probably because the for loop makes the code writing process faster, and the end result more readable. This is especially the case in block-based graphical programming environments since the blocks are very verbose, quickly cluttering up the workspace. Consequently, we believe that the learners in the *create* group see more value in the usage of a loop. For them using loops makes the coding process faster and more clear. This is in contrast to the *fix* group, which was required to use loops in some assignments to make their code more clear and concise but saw less value in them since they were never required to add a lot of code themselves,

Table 11
Logging data results for the create and fix groups. The values for condition/selection, for-loop and time-delay are represented as number of times they occurred in the code when a code change occurs.

	Create	**Fix**
Blocks created	8772	2769
Code changes	52401	33834
Condition/selection	6255(11.9%)	5429(16.0%)
For-loop	14575(27.8%)	5173(15.3%)
Time-delay	34698(66.2%)	29667(87.7%)

only modify the incorrect code they got.

Looking at the different elements in the experiential dimension, it is clear that the learners in the *create* group performed better than the ones in the *fix* group. Moreover, the students in the *create* group were more engaged and showed less negative emotions. This stands in contrast to the results for the intrinsic dimension, where we did not observe any differences between the groups. In the discussion section, we elaborate on the differences between the two groups and link all evaluation metrics together.

Discussion and conclusions

Figure 4 visualizes the global comparison between the two workshops. We generated this figure by selecting indicators from the different assessed dimensions, normalizing them (by dividing the score on one dimension by the sum of the scores for both dimensions), and plotting them in a spider web plot. The following dimensions are visualized: total score on the attitude test, number of programming concepts used by the videotaped students, number of computational thinking concepts used by the videotaped students, number of times engagement was observed in the group of taped students, the average score on both programming tests, the average score on both computational thinking tests and the number of code changes. Clearly, the *create* workshop scores better than the *fix* workshop. Only the scores on the attitude test are at the same level.

Our research question focused on analyzing the differences between the *create* and *fix* methods from the perspective of learning ecology. To assess the experiential dimension, we looked at both learning achievement as well as learning experience. Looking at the differences in learning achievement, it can be observed that, on our set of programming questions, the learners in the *fix* group scored significantly lower than the ones in the *create* group (Table 10). This is surprising since literature indicates that teaching programming through fixing should not have negative effects on programming knowledge acquisition (Lee et al., 2014). Moreover, code completion strategies, similar to the ones in our *fix* workshop also should not negatively affect knowledge acquisition (Chang et al., 2000; Garner, 2009; Van Merriënboer & De Croock, 1992). We suspect this negative effect on programming knowledge acquisition is a result of a disproportionate cognitive load. Previous research has shown that reading and understanding code is a different and more difficult skill than writing code (Lister et al., 2009). Moreover, others have shown that young learners experience

debugging as intellectually challenging (Liu et al., 2017). This challenging environment can lead to extraneous cognitive load resulting in lower learning efficiency, higher frustration and fewer engagement (Kester et al., 2006).

The idea that the *fix* method induces a higher cognitive load is supported by our comparison of the learning experiences during the workshop. Looking at the interactions of the learners (page 16), some subtle differences arise. Learners in the *fix* group had more difficulties solving the assignments, perform less code interactions, use multiple different programming concepts less often, and are less engaged while coding than the learners in the *create* group (Table 7 and Table 9). These difficulties resulted in more anger and frustration during the workshop leading to a loss in motivation. These observations are confirmed by the results of our emotion coding scheme (Table 9) which shows that learners in the *create* group show less negative emotions, more positive emotions and are more engaged than the learners in the *fix* group. These results are backed up by our programming data analysis. The learners in the *create* group create more blocks, change the code more often, and use different coding concepts more often. Most likely this is because the learners in the *fix* group spend more time reading and understanding the code than actually modifying it.

Even though teaching programming through fixing should reduce the cognitive load by providing a large part of the program, limiting the space of possibilities, and having a clear goal learners have to work towards (Garner, 2009), it seems that this reduced load is not offset by the additional intellectual challenges introduced by having to find the specific location of the error and figuring out how to fix it. It seems like there is a delicate balance between the extra scaffolding the fixing strategy provides and the additional cognitive challenges it imposes. In our experiment, these extra cognitive challenges take the upper hand resulting in a higher cognitive load. This increased cognitive load leads to more frustration (Kester et al., 2006), which again leads to lower performance. Some argue that, because teaching programming through debugging is more cognitively challenging and requires multiple types of computational thinking skills, it is better for supporting the acquisition of higher-order thinking skills (Klahr & Carver, 1988; Román-González et al., 2017). Even though this might be the case for more extensive programming courses, we were not able to confirm this. Table 10 shows no significant differences in scores between the groups on the computational thinking test.

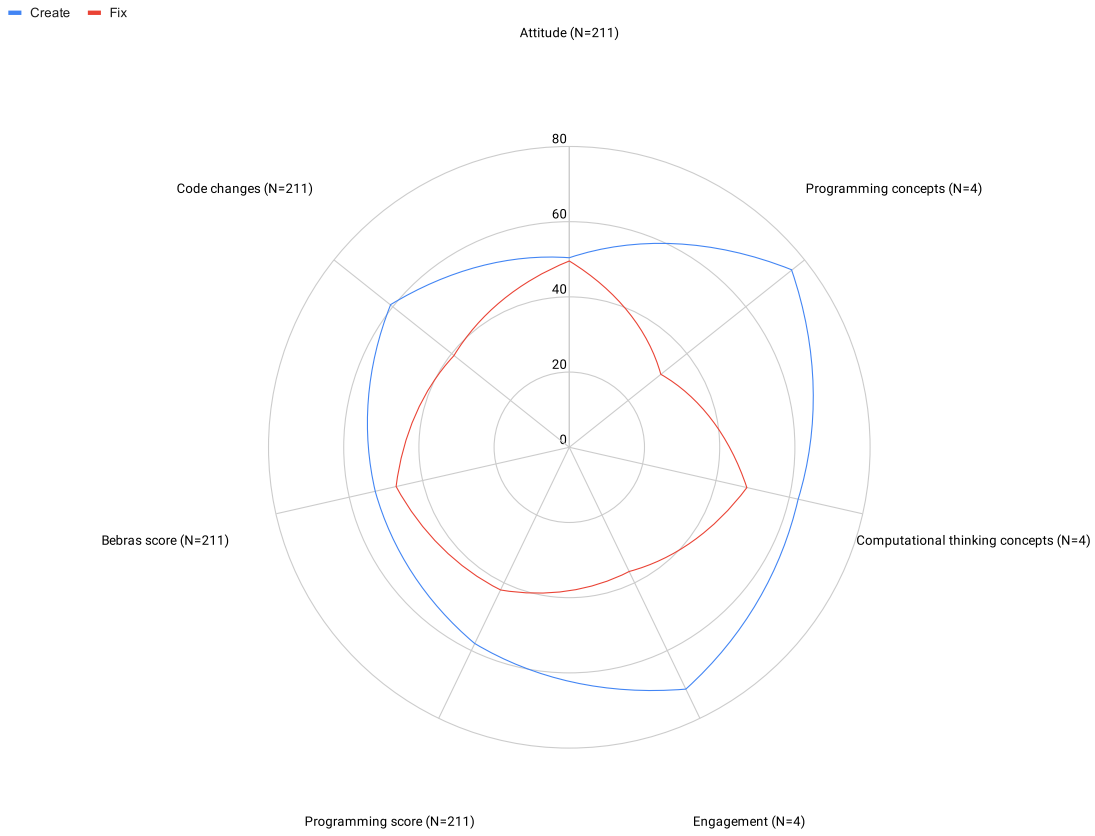
We are aware that our results highly depend on the instructional design, changing aspects of this design might yield different results. When teachers have more time available to teach programming it is plausible that including debugging exercises might be beneficial. However, determining the proper mix of creating and debugging exercises requires further research. Another possible drawback of our analysis is the limited information we collected about the experiential dimension of learning ecology. To establish a valid baseline for the comparison between the two teaching methods we analyzed, we used an attitude test as well as an analysis of the learners in our qualitative study (Table 6 and page 15). Even though our results show no noticeable differences between the groups, the techniques we used only provide a limited view of the actual dispositions of the learners. Attitude provides only a momentary image of the learners underlying dispositions. Additionally, other dispositions like prior knowledge or family context were not considered. However, we do believe that the aspects we have analyzed strengthen our assumption that there was no bias between our two experimental groups. A third important limitation of our study is that, even though our

workshop included one and a half days of learning, the overall scores on the programming tests are still low. This indicates that our workshop was not very effective at improving the comprehension of programming concepts in both groups. This shows that it might not be feasible to have the learners in this age group acquire a sufficient understanding of the programming concepts taught in our workshop in a limited time. Nevertheless, this does not invalidate the differences between the two groups. The fix group does learn significantly less than the create group.

Based on our results we can formulate some concrete guidelines for teachers helping them to improve the integration of programming into their curricula. Firstly, when choosing a set of programming exercises, the teacher should be aware of the additional cognitive load fixing-problems impose. Consequently, limiting those problems is important. We do not suggest not using these kinds of problems since they can be an important tool to provide faster learners with extra challenges. Secondly, the teaching method chosen has a limited effect on computational thinking skills. When computational thinking is a goal, teachers should select teaching techniques proven to be beneficial for the improvement of computational thinking and not purely only rely on programming problems. Thirdly, teachers should be aware that specific types of problems can affect negative emotions in students leading to less engagement. This should be avoided as much as possible to limit long-term negative effects. Finally, primary school teachers should be aware that letting learners freely explore programming can lead to a more positive programming experience and a better understanding of programming concepts. Currently, some teachers prefer ready-to-use scaffolded programming assignments defining a clear context for teaching, assessment, and grading. Our results should encourage teachers to explore more open teaching environments allowing more exploration.

Overall, our study has shown that integrating programming into a realistic primary robotics workshop is best done using the *create* method. It gives learners the freedom to explore different programming concepts at their own pace. This gives them more insight into the programming concepts and shows the value of concepts like iteration, which facilitates the learning process. This method takes a more gentle approach when introducing programming resulting in a more pleasant learning process with less negative emotions and more engagement. This pleasant learning experience also results in better test scores.

Figure 4. Scores on the analyzed dimensions for both the *create* and *fix* workshops. The following dimensions are first normalized and then visualized: total score on the attitude test, number of programming concepts used by the taped students, number of computational thinking concepts used by the taped students, number of times engagement was observed in the group of taped students, the average score on both programming tests, the average score on both computational thinking tests and the number of code changes.



References

Abdul-Rahman, S.-S. & Du Boulay, B. (2014). Learning programming via worked-examples: Relation of learning styles to cognitive load. *Computers in Human Behavior*, *30*, 286–298.

Ahmadzadeh, M., Elliman, D. & Higgins, C. (2005). An analysis of patterns of debugging among novice computer science students. *Proceedings of the 10th annual SIGCSE conference on Innovation and technology in computer science education*, 84–88.

Barendsen, E., Mannila, L., Demo, B., Grgurina, N., Izu, C., Mirolo, C., Sentance, S., Settle, A. & Stupurien, G. (2015). Concepts in k-9 computer science education. *Proceedings of the 2015 ITiCSE on working group reports*, 85–116.

Bau, D., Gray, J., Kelleher, C., Sheldon, J. & Turbak, F. (2017). Learnable programming: Blocks and beyond. *Communications of the ACM*, *60*(6), 72–80.

Bell, T., Alexander, J., Freeman, I. & Grimley, M. (2009). Computer science unplugged: School students doing real computing without computers. *The New Zealand Journal of Applied Computing and Information Technology*, *13*(1), 20–29.

Bosch, N., D'Mello, S. & Mills, C. (2013). What emotions do novices experience during their first computer programming learning session? *International Conference on Artificial Intelligence in Education*, 11–20.

Brennan, K. & Resnick, M. (2012). New frameworks for studying and assessing the development of computational thinking. *Proceedings of the 2012 annual meeting of the American Educational Research Association, Vancouver, Canada*, *1*, 25.

Celepkolu, M. & Boyer, K. E. (2018). Thematic analysis of students' reflections on pair programming in cs1. *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*, 771–776.

Chang, K.-E., Chiao, B.-C., Chen, S.-W. & Hsiao, R.-S. (2000). A programming learning system for beginners-a completion strategy approach. *IEEE Transactions on Education*, *43*(2), 211–220.

Chen, G., Shen, J., Barth-Cohen, L., Jiang, S., Huang, X. & Eltoukhy, M. (2017). Assessing elementary students' computational thinking in everyday reasoning and robotics programming. *Computers & Education*, *109*, 162–175.

Cobb, P., Confrey, J., DiSessa, A., Lehrer, R. & Schauble, L. (2003). Design experiments in educational research. *Educational researcher*, *32*(1), 9–13.

Dagiene, V. & Futschek, G. (2008). Bebras international contest on informatics and computer literacy: Criteria for good tasks. *International Conference on Informatics in Secondary Schools-Evolution and Perspectives*, 19–30.

Dagiene, V. & Sentance, S. (2016). It's computational thinking! bebras tasks in the curriculum. *International Conference on Informatics in Schools: Situation, Evolution, and Perspectives*, 28–39.

Du, Y., Luxton-Reilly, A. & Denny, P. (2020). A review of research on parsons problems. *Proceedings of the Twenty-Second Australasian Computing Education Conference*, 195–202.

Else-Quest, N. M., Hyde, J. S. & Hejmadi, A. (2008). Mother and child emotions during mathematics homework. *Mathematical Thinking and Learning*, *10*(1), 5–35.

Ericson, B. J., Margulieux, L. E. & Rick, J. (2017). Solving parsons problems versus fixing and writing code. *Proceedings of the 17th Koli Calling International Conference on Computing Education Research*, 20–29.

Feurzeig, W. et al. (1969). Programming-languages as a conceptual framework for teaching mathematics. final report on the first fifteen months of the logo project. *ACM SIGCUE Outlook*.

Garner, S. (2009). A quantitative study of a software tool that supports a part-complete solution method on learning outcomes. *Journal of Information Technology Education: Research*, *8*(1), 285–310.

González-Sanmamed, M., Muñoz-Carril, P.-C. & Santos-Caamaño, F.-J. (2019). Key components of learning ecologies: A delphi assessment. *British Journal of Educational Technology*.

Hsu, T.-C., Chang, S.-C. & Hung, Y.-T. (2018). How to learn and how to teach computational thinking: Suggestions based on a review of the literature. *Computers & Education*, *126*, 296–310.

Jackson, N. J. (2013). The concept of learning ecologies. *Lifewide learning, education and personal development e-book*.

Kazakoff, E. R., Sullivan, A. & Bers, M. U. (2013). The effect of a classroom-based intensive robotics and programming workshop on sequencing ability in early childhood. *Early Childhood Education Journal*, *41*(4), 245–255.

Kester, L., Lehnen, C., Van Gerven, P. W. & Kirschner, P. A. (2006). Just-in-time, schematic supportive information presentation during cognitive skill acquisition. *Computers in Human Behavior*, *22*(1), 93–112.

Kim, C., Yuan, J., Vasconcelos, L., Shin, M. & Hill, R. B. (2018). Debugging during block-based programming. *Instructional Science*, *46*(5), 767–787.

Klahr, D. & Carver, S. M. (1988). Cognitive objectives in a logo debugging curriculum: Instruction, learning, and transfer. *Cognitive Psychology*, *20*(3), 362–404.

Koch, S. C. & Zumbach, J. (2002). The use of video analysis software in behavior observation research: Interaction patterns in task-oriented small groups. *Forum Qualitative Sozialforschung/Forum: Qualitative Social Research*, *3*.

Lee, M. J., Bahmani, F., Kwan, I., LaFerte, J., Charters, P., Horvath, A., Luor, F., Cao, J., Law, C., Beswetherick, M. et al. (2014). Principles of a debugging-first puzzle game for computing education. *Visual Languages and Human-Centric Computing (VL/HCC), 2014 IEEE Symposium on*, 57–64.

Lewis, M., Haviland-Jones, J. M. & Barrett, L. F. (2010). *Handbook of emotions*. Guilford Press.

Lister, R., Fidge, C. & Teague, D. (2009). Further evidence of a relationship between explaining, tracing and writing skills in introductory programming. *Acm sigcse bulletin*, *41*(3), 161–165.

Liu, Z., Zhi, R., Hicks, A. & Barnes, T. (2017). Understanding problem solving behavior of 6–8 graders in a debugging game. *Computer Science Education*, *27*(1), 1–29.

Luckin, R. (2008). The learner centric ecology of resources: A framework for using technology to scaffold learning. *Computers & Education*, *50*(2), 449–462.

Luxton-Reilly, A., Albluwi, I., Becker, B. A., Giannakos, M., Kumar, A. N., Ott, L., Paterson, J., Scott, M. J., Sheard, J. & Szabo, C. (2018). Introductory programming:

A systematic literature review. *Proceedings Companion of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education*, 55–106.

Lye, S. Y. & Koh, J. H. L. (2014). Review on teaching and learning of computational thinking through programming: What is next for k-12? *Computers in Human Behavior*, *41*, 51–61.

Malik, S. I. & Coldwell-Neilson, J. (2017). A model for teaching an introductory programming course using adri. *Education and Information Technologies*, *22*(3), 1089–1120.

McCauley, R., Fitzgerald, S., Lewandowski, G., Murphy, L., Simon, B., Thomas, L. & Zander, C. (2008). Debugging: A review of the literature from an educational perspective. *Computer Science Education*, *18*(2), 67–92.

McDowell, C., Hanks, B. & Werner, L. (2003). Experimenting with pair programming in the classroom. *Proceedings of the 8th annual conference on Innovation and technology in computer science education*, 60–64.

Plass, J. L. & Kalyuga, S. (2019). Four ways of considering emotion in cognitive load theory. *Educational Psychology Review*, *31*(2), 339–359.

Popat, S. & Starkey, L. (2019). Learning to code or coding to learn? a systematic review. *Computers & Education*, *128*, 365–376.

Rodrigo, M. M. T. & Baker, R. S. (2009). Coarse-grained detection of student frustration in an introductory programming course. *Proceedings of the fifth international workshop on Computing education research workshop*, 75–80.

Román-González, M., Moreno-León, J. & Robles, G. (2017). Complementary tools for computational thinking assessment. *Proceedings of International Conference on Computational Thinking Education (CTE 2017), S. C Kong, J Sheldon, and K. Y Li (Eds.). The Education University of Hong Kong*, 154–159.

Shaffer, D., Doube, W. & Tuovinen, J. (2003). Applying cognitive load theory to computer science education. *PPIG*, 12.

Shaffer, D. W. & Resnick, M. (1999). " thick" authenticity: New media and authentic learning. *Journal of interactive learning research*, *10*(2), 195–216.

Summers, R. & Abd-El-Khalick, F. (2018). Development and validation of an instrument to assess student attitudes toward science across grades 5 through 10. *Journal of Research in Science Teaching*, *55*(2), 172–205.

Sweller, J. (2011). Cognitive load theory. *Psychology of learning and motivation* (pp. 37–76). Elsevier.

Sweller, J. (2020). Cognitive load theory and educational technology. *Educational Technology Research and Development*, *68*(1), 1–16.

Van Merrienboer, J. J. & Sweller, J. (2005). Cognitive load theory and complex learning: Recent developments and future directions. *Educational psychology review*, *17*(2), 147–177.

Van Merriënboer, J. J. & De Croock, M. B. (1992). Strategies for computer-based programming instruction: Program completion vs. program generation. *Journal of Educational Computing Research*, *8*(3), 365–394.

Vasilopoulos, I. V. & Van Schaik, P. (2019). Koios: Design, development, and evaluation of an educational visual tool for greek novice programmers. *Journal of Educational Computing Research*, *57*(5), 1227–1259.

Vlaanderen, O. (1997). Onderwijsdoelen basisonderwijs vlaanderen [1997 (accessed 2019-12-09)]. https://onderwijsdoelen.be/resultaten?intro=basisonderwijs

Vlaanderen, O. (2019). Onderwijsbegroting [2019 (accessed 2019-12-09)]. statistiekvlaanderen.be/nl/onderwijsbegroting

Weintrop, D. & Wilensky, U. (2015). To block or not to block, that is the question: Students' perceptions of blocks-based programming. *Proceedings of the 14th international conference on interaction design and children*, 199–208.

Wing, J. M. (2006). Computational thinking. *Communications of the ACM*, *49*(3), 33–35.

Witherspoon, E. B., Higashi, R. M., Schunn, C. D., Baehr, E. C. & Shoop, R. (2017). Developing computational thinking through a virtual robotics programming curriculum. *ACM Transactions on Computing Education (TOCE)*, *18*(1), 1–20.

Zhong, B., Wang, Q. & Chen, J. (2016). The impact of social factors on pair programming in a primary school. *Computers in Human Behavior*, *64*, 423–431.

Appendix

Appendix A

- **Session 1**

  1. The learners know that a program is a sequence of instructions that are accurate and in the right order;
  2. The learners write, change, run, upload code in the DwenguinoBlockly environment;
  3. The learners know how to connect the microcontroller board to the computer;
  4. The learners print text to the lcd-screen;
  5. The learners know the difference between setup and loop;
  6. The learners correctly use the wait-block;
  7. The learners use counting loops and can identify when and why they are useful;
  8. The learners control LEDs and the buzzer on the microcontroller board;
  9. The learners convert microseconds into seconds;

- **Session 2**

  1. The learners construct a two-wheeled riding robot using the Dwengo robot kit:
     (a) The learners connect the structural parts using nuts and bolts;
     (b) The learners connect the battery using a rubber band;
     (c) The learners connect the motors the microcontroller board using a screwdriver;
  2. The learners use the DC-motor block to make their robot ride;
  3. The learners run their program inside the simulator;
  4. The learners use the wait-block to add time-based behavior to their robot;
  5. The learners make their robot drive into different shapes, line, square, circle;

- **Session 3**

  1. The learners connect a sonar distance sensor to their robot;
  2. The learners understand the function of a sensor;
  3. The learners evaluate conditionals and perform different actions when they are true or false;
  4. The learners use an if statement with the condition to read the distance from a sonar sensor;
  5. The learners program their robot to act on sensor input;
  6. The learners understand the effect of the real world on the programming process;